What I made:

Dancing links implementation of algorithm X with an altered version to allow partial solutions.

How it works:

It creates a sparse matrix that represents all the possible ways that pieces can be placed. All the places that would be filled in by an option are represented by nodes. These nodes link to their neighbors and header nodes. Header nodes are special nodes that are referred to by every node in that column and contain some extra information.

The algorithm than tries to fill the container as much as possible by picking items to fill the space. Each time it picks an item it will remove overlapping items from the items that would still be left. It will keep doing this until there are no items left that could be fitted in. It than stores this result for later use. The order in which the items that are going to fill the container is also important in the case of exact covers, since items that fill it up quicker are more likely to reduce the amount of steps needed to fill in the rest and thus reduce the amount of backtracking needed. The backtracking occurs whenever no more items can be used to fill in the container.

In the case of exact cover, once the algorithm is done it will either have found a way to fill in the entire space or it will be certain that there’s no way to fill it in complete. In the case of optimizing the score, the program will go trough all the saved (partial-)solutions and determine their score. The score is created based on the amount of items used and their value. The solution with the highest score is than picked.

Challenges:

In order to use dancing links an input array of all possible ways the items can be placed is needed. The creation of this array was rather difficult because of all the possible rotations in 3D can be applied to the pentominoes.

The creation of the dancing links structure itself was also rather difficult, since it requires a deep understanding of how it works and involves a lot of detailed steps.

Dancing links isn’t meant to solve partial cover problems, so in order to allow it to still be used for this purpose a few adjustments were needed. This was accomplished by saving the state of the (partial-)solution whenever no new items could be placed. Since this will always be the case at an endpoint of the search tree, all these endpoints can be compared based on the score that this solution has. The best score than be chosen. It also filters out any locations that can never be filled from the input, this prevents it from not being able to reach a easily detectable endpoint.

Sources used:

<https://www.geeksforgeeks.org/exact-cover-problem-algorithm-x-set-2-implementation-dlx/>

<https://stackoverflow.com/questions/39939710/java-how-to-implement-dancing-links-algorithm-with-doublylinkedlists>

<https://github.com/benfowler/dancing-links/tree/master/dlx/src>

<https://github.com/Warren-Partridge/algorithm-x/blob/master/DancingLinks.java>

<https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/sudoku.paper.html>

<https://github.com/benfowler/dancing-links/blob/master/dlx/src/main/java/au/id/bjf/dlx/DLX.java>

<http://sudopedia.enjoysudoku.com/Dancing_Links.html>

The code itself is also partially based on a combination of example code (listed above) in order to get a basic version working.

Neat images

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated